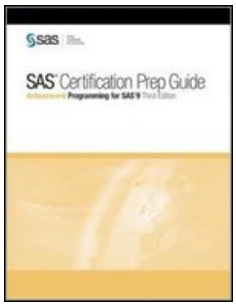


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 24: Querying Data Efficiently

Overview

Introduction

SAS provides a variety of techniques for querying data that enable you to create the results that you want in different ways. In this chapter, you learn to select the most efficient query techniques from those listed below, based on comparisons of resource usage.

Task	Techniques
selecting a subset	▪ WHERE statement that references a data set that has been indexed
creating a detail report	▪ PRINT procedure ▪ SQL procedure
creating a summary report for one class variable or a combination of class variables	▪ MEANS procedure (or SUMMARY procedure) ▪ TABULATE procedure ▪ REPORT procedure ▪ SQL procedure ▪ DATA step

Note This chapter does not cover the SAS Scalable Performance Data Engine (SAS SPD Engine). For details about using the SAS SPD Engine to improve performance, see the SAS documentation.

Objectives

In this chapter, you learn to

- identify the costs and benefits of using an index
- identify the factors that affect whether SAS uses an index for WHERE processing
- determine whether SAS is likely to use an index to process a particular WHERE expression
- identify the main features of compound optimization
- identify the effect of indexing and order of data on WHERE processing
- print centile information for a data file
- identify the relative efficiency of the PRINT procedure and the SQL procedure for creating detail reports
- identify the relative efficiency of five tools for summarizing data for one categorical variable
- identify the relative efficiency of three ways of using the MEANS procedure to summarize data for selected combinations of categorical variables.

Prerequisites

Before beginning this chapter, you should complete the following chapters:

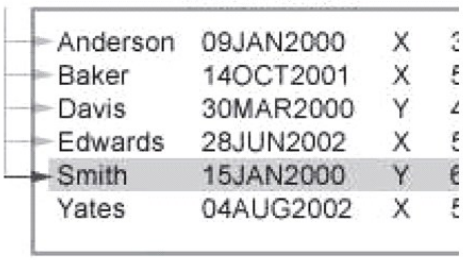
- Part 1: SQL Processing with SAS
  - "Performing Queries Using PROC SQL" on page 4
  - "Performing Advanced Queries Using PROC SQL" on page 29

Using an Index for Efficient WHERE Processing

Overview

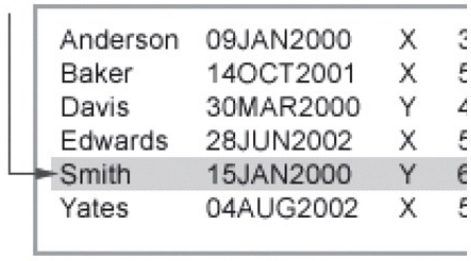
When processing a WHERE expression, SAS determines which of the following access methods is likely to be most efficient:

Sequential access



SAS searches through all observations sequentially (in the order in which they are stored in the data file).

Direct access



SAS uses an *index* to access specific observations directly. Using an index to process a WHERE expression is referred to as *optimizing* the WHERE expression.

Using an index to process a WHERE expression improves performance in some situations but not in others. For example, it is more efficient to use an index to select a small subset than a large subset. In addition, an index conserves some resources at the expense of others.

By deciding whether to create an index, you also play a role in determining which access method SAS can use. When your program contains a WHERE expression, you should determine which access method is likely to be more efficient. If direct access is likely to be more efficient, you can make sure that an index is available by creating a new index or by maintaining an existing index.

To help you make a more effective decision about whether to create an index, this topic and the next few topics provide you with a closer look at the following:

- steps that SAS performs for sequential access and direct access
- benefits and costs of index usage
- steps that SAS performs to determine which access method is most efficient
- factors affecting resource usage for indexed access
- guidelines for deciding whether to create, use, and maintain an index.

Note You should already know how to create and maintain indexes by using the INDEX= data set option in the DATA statement, the DATASETS procedure, and the SQL procedure. To review these SAS elements, see "Creating and Managing Indexes Using PROC SQL" on page 238 and "Creating Samples and Indexes" on page 470.

Note SAS can also use an index to process a BY statement. BY processing enables you to process observations in a specific order according to the values of one or more variables that are specified in a BY statement. Indexing a data file enables you to use a BY statement without sorting the data file. When you specify a BY statement, SAS checks the value of the Sorted indicator. If the Sorted indicator is set to NO, then SAS looks for an appropriate index. If an appropriate index exists, the software automatically retrieves the observations from the data file in indexed order. Using an index to process a BY statement might not always be more efficient than simply sorting the data file. Therefore, using an index for a BY statement is generally for convenience, not for performance.

Accessing Data Sequentially

When accessing observations sequentially, SAS must search through all observations in the order in which they are stored

in the data file.

Example

Suppose you want to create a new data set, *Company.D02jul2000*, that contains a subset of observations from the data set *Company.Dates*. The following DATA step uses a WHERE statement to select all observations in which the value of **`Date_ID`** is *02JUL2000-*.

```
data company.d02jul2000;
   set company.dates;
   where date_id='02JUL2000'd;
run;
```

The data set *Company.Dates* does *not* have an index defined on the variable **`Date_ID`**, so SAS must use *sequential access* to process the WHERE statement.

Note If the data set **`company.dates`** has been sorted by **`date_id`**, SAS searches the data sequentially until the WHERE criterion **`(date_id='02JUL2 000'd)`** has been satisfied.

Accessing Data Directly

When using an index for WHERE processing, SAS goes straight to each observation that contains the value without having to read every observation in the data set.

Example

Suppose you have defined an index on the variable **`Date_ID`** in the *Company.Dates* data set. This time, when you submit the following DATA step, SAS *uses the index* to process the WHERE statement:

```
data company.d02jul2000;
   set company.dates;
   where date_id='02JUL2000'd;
run;
```

The process of retrieving data via an index (direct access) is more complicated than sequentially processing data, so direct access requires more CPU time per observation retrieved than sequential access. However, for a small subset, using an index can decrease the number of pages that SAS has to load into input buffers, which reduces the number of I/O operations.

Note When the values in the data set are sorted in the order in which they occur in the index, the qualified observations are adjacent to each other. In this situation, SAS loads fewer pages into the input buffer than If the data is randomly distributed throughout the data set. Therefore, fewer I/O operations are required when the data set is sorted. However, there is a greater chance that SAS will need to load the same page of data multiple times, and that more I/O operations will be required, when the data values are distributed randomly and more than one value needs to be selected to satisfy the WHERE statement (using an operator other than the equals operator).

Benefits and Costs of Using an Index

As the preceding examples show, both benefits and costs are associated with using an index. Weighing these benefits and costs is an important part of deciding whether using an index is efficient.

The main *benefits* of using an index include the following:

- provides fast access to a small subset of observations
- returns values in sorted order
- can enforce uniqueness.

The main *costs* of using an index include the following:

- requires extra CPU cycles and I/O operations for creating and maintaining an index
- requires increased CPU time and I/O activity for reading the data

- requires extra disk space for storing the index file
- requires extra memory for loading index pages and extra code for using the index.

Note SAS requires additional buffers when an index file is used. When a data file is opened, SAS opens the index file, but not the indexes. Buffers are not required unless SAS uses an index, but SAS allocates the buffers to prepare for using the index. The number of levels of an index determines the number of buffers that are allocated. The maximum number of buffers is three for data files that are open for input; the maximum number is four for data that is open for update. These buffers can be used for other processing If they are not used for indexes.

How SAS Selects an Access Method

When SAS processes a WHERE expression, it first determines whether to use direct access or sequential access by performing the following steps:

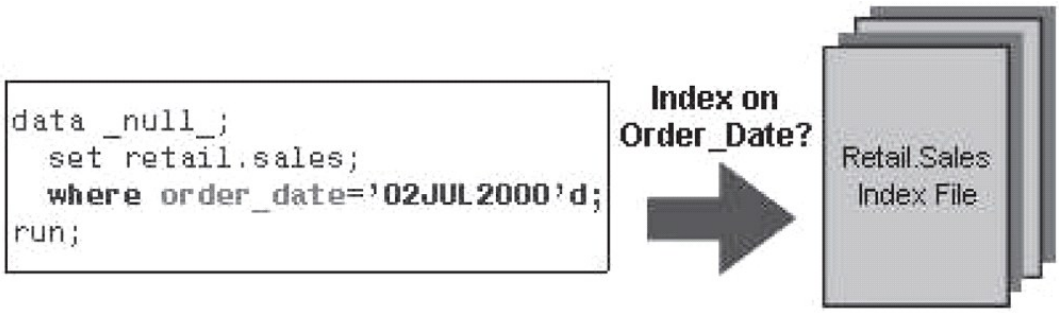
1. identifies available indexes
2. identifies conditions that can be optimized
3. estimates the number of observations that qualify
4. compares probable resource usage for both methods.

In the next few sections, each step of this process is explained in detail.

Identifying Available Indexes

Overview

The first step for SAS is to *determine whether there are any existing indexes* that might be used to process the WHERE expression. Specifically, SAS checks the variable in each condition in the WHERE expression to determine whether the variable is a *key variable* in an index.



SAS can use either a simple index or a composite index to optimize a WHERE expression. To be considered for use in optimizing a single WHERE condition, one of the following requirements must be met:

- the variable in the WHERE condition is the *key variable* in a *simple* index
- the variable in the WHERE condition is the *first key variable* in a *composite* index.

SAS identifies all indexes that are defined on any variable in the WHERE expression. However, no matter how many indexes are available, *SAS can use only one index to process a WHERE expression*. So, if multiple indexes are available, SAS must choose between them.

When SAS looks for available indexes, there are three possible outcomes:

If…	Then…
there is *no index* defined on any variables in the WHERE expression	SAS *does not continue* with the decision process. SAS must use sequential access to process the WHERE expression.
there is *one available index* that is defined on	SAS *continues* with the decision process and determines whether using the

one or more variables in the WHERE expression	available index is more efficient than using sequential access.
there are *multiple available indexes*, each of which is defined on one or more of the variables in the WHERE expression	SAS *continues* with the decision process. SAS must choose between the available indexes in the next few steps. SAS tries to select the index that satisfies the most conditions and that selects the smallest subset of observations.

Note If a program specifies both a WHERE expression and a BY statement, SAS looks for one index that satisfies conditions for both. If such an index is not found, the BY statement takes precedence so that SAS can ensure that the data is returned in sorted order. With a BY statement, SAS cannot use an index to optimize a WHERE expression If the optimization invalidates the BY order.

Example: Identifying One Available Index

Suppose you submit a program that contains the following WHERE statement, and suppose that the data set has one index, as shown below:

WHERE Statement	Available Index
`where delivery_date='02jul2000'd`	simple index defined on **`Delivery_Date`**

This WHERE expression has one condition, and the variable in that condition **`(Delivery_Date)`** is the key variable in the simple index. If all other requirements for optimization are met in later steps, then SAS can use this index to optimize the WHERE expression.

Likewise, If the only available index is a composite index in which **`Delivery_Date`** is the first key variable, then SAS can use the index if all other requirements for optimization are met.

Even if a WHERE statement has multiple conditions, SAS can use either a simple index or a composite index to optimize just one of the conditions. For example, suppose your program contains a WHERE statement that has two conditions, and suppose that the data set has one index, as shown below:

WHERE Statement	Available Index
`where order date='01jan2000'd and delivery_date='02jul2000'd';`	simple index defined on **`Delivery_Date`**

Assuming that all other requirements for optimization are met, SAS can use this index to optimize the second condition in this WHERE expression.

Example: Identifying Multiple Available Indexes

Suppose your program contains a WHERE statement with two conditions, and suppose that each condition references a key variable in a different index, as shown below:

WHERE Statement	Available Index
`where order_date='01jan2000'd and delivery_date='02jul2000'd';`	▪ simple index defined on **`Order_Date`** ▪ simple index defined on **`Delivery_Date.`**

Although two indexes are available, SAS can use only one index to optimize a WHERE statement. In a later step of the process, SAS will try to select the index that satisfies the most conditions and that selects the smallest subset of observations.

Compound Optimization

SAS usually uses an index to process just one condition, no matter how many conditions and variables a WHERE expression contains. However, in a process called *compound optimization*, SAS can use a composite index to optimize multiple conditions on multiple variables, which are joined with a logical operator such as AND. Constructing your WHERE expression to take advantage of multiple key variables in a single index can greatly improve performance.

In order for compound optimization to occur, at least the first two key variables in the composite index must be used in the WHERE conditions. Later in this chapter, you will learn about other requirements that must be met in order for compound optimization to occur.

Note The WHERE expression can also contain non-indexed variables, and the key variables and non-indexed variables can appear in any order in the expression.

Example: Composite Index That Can Be Used to Optimize Multiple Conditions

Suppose your program contains a WHERE statement that has two conditions, and suppose that each condition references one of the first two key variables in a composite index:

WHERE Statement	Available Index
`where order_date='01jan2000'd and delivery_date='02jul2000'd';`	composite index defined on the following variables: ▪ **Order_Date** (first key variable) ▪ **Delivery_Date** (second key variable) ▪ **Product_ID** (third key variable)

Because the two variables that are referenced in the WHERE expression are the first two key variables in the composite index, SAS can use the composite index for compound optimization If the WHERE conditions meet all other requirements for optimization.

Example: Composite Index That Can Be Used to Optimize One Condition

The following WHERE statement also contains two conditions, and each condition references one of the variables in the composite index:

WHERE Statement	Available Index
`where order_date='01jan2000'd and product_id='220101400106';`	composite index defined on the following variables: ▪ **Order_Date** (first key variable) ▪ **Delivery_Date** (second key variable) ▪ **Product_ID** (third key variable)

As in the previous WHERE statement, **Order_Date** is the first key variable in the index. However, in this situation, the composite index can be used to optimize only the first condition. The second condition references the third key variable, **Product_ID**, but the WHERE expression does not reference the second key variable, **Delivery_Date.** Without a reference to both the first and second key variables, compound optimization cannot occur.

Example: Composite Index That Cannot Be Used for Optimizing

Now suppose your program contains a WHERE statement that references only the second and third key variables in the composite index, as shown below:

WHERE Statement	Available Index
`where delivery_date='02jul2000'd' and product_id='220101400106';`	composite index defined on the following variables: ▪ **Order_Date** (first key variable) ▪ **Delivery_Date** (second key variable) ▪ **Product_ID** (third key variable)

In this situation, SAS *cannot* use the index for optimization at all because the WHERE statement does not reference the first key variable.

Identifying Conditions That Can Be Optimized

In addition to containing key variables, WHERE conditions must meet other requirements in order to be candidates for optimization. SAS considers using an index only for WHERE conditions that contain *certain operators and functions*. Therefore, the next step for SAS is to consider the operators and functions in the conditions that contain key variables.

Requirements for Optimizing a Single WHERE Condition

SAS considers using an index for a WHERE condition that contains any of the following operators and functions:

Note For all of the following examples, assume that the data set has simple indexes on the variables **`Quarter`**, **`Date_ID`**, and **`Region`**.

Operator	Example

comparison operators and the IN operator	`where quarter = '1998Q1';`
	`where date_id < '03JUL2000'd;`
	`where quarter in ('1998Q2','1998Q3')`

comparison operators with NOT	`where quarter ne '1999Q1';` `where quarter not in ('1999Q1','1999Q4')`
comparison operators with the colon modifier You can add a colon modifier (:) to any comparison operator to compare only a specified prefix of a character string. The colon modifier cannot be used with PROC SQL; use the LIKE operator instead.	`where quarter =: '1998';`
CONTAINS operator	`where quarter contains 'Q4';`

fully bounded range conditions that specify both an upper and lower limit, which includes the BETWEEN-AND operator	`where '01Jan1999'd < date_id`
	`< '31Dec1999'd;`
	`where date_id between '01Jan1999'd`
	`and '31Dec1999'd`

pattern-matching operator LIKE	`where quarter like '%Q%';`
IS NULL or IS MISSING operator	`where quarter is null;` `where quarter is missing;`

Function	Example
TRIM function	`where trim(region) = 'Queensland';`
SUBSTR function in the form of **WHERE** SUBSTR *(variable,position,length)='string';* with these conditions: ▪ position = 1 ▪ *length* is less than or equal to the length of variable ▪ *length* is equal to the length of the string.	`where substr(quarter,1,4) = '1998';`

Caution Most but not all of the requirements listed above also apply to compound optimization. Requirements for compound optimization are covered later in this chapter.

WHERE Conditions That Cannot Be Optimized

SAS does *not* use an index to process a WHERE condition that contains any of the elements listed below.

Note For all of the following examples, assume that the data set has simple indexes on the variables **Date_ID, Quarter**, and **Quantity.**

Element in WHERE Condition	Example
any function other than TRIM or SUBSTR	`where weekday(date id)=2;`
a SUBSTR function that searches a string beginning at any position after the first	`where substr(quarter,6 ,1)='1';`
the sounds-like operator (=*)	`where quarte]=*'1900Q0';`
arithmetic operators	`where quantity=quantity+1;`
a variable-to-variable condition	`where quantity gt threshold;`

Requirements for Compound Optimization

Most of the same operators that are acceptable for optimizing a single condition are also acceptable for compound optimization. However, compound optimization has special requirements for the operators that appear in the WHERE expression:

- The WHERE conditions must be connected by using either the AND operator or, if all conditions refer to the same variable, the OR operator.
- At least one of the WHERE conditions that contains a key variable must contain the EQ or IN operator.

Example: Compound Optimization

Suppose your program contains the following WHERE statement, which selects all people whose name is John Smith. The WHERE statement contains two conditions, each of which references a different variable:

```
where lastname eq 'Smith' and
      frstname eq 'John';
```

Suppose **Lastname** is the first key variable and **Frstname** is the second key variable in a compound index. This WHERE statement meets all requirements for compound optimization:

- The WHERE expression references at least the first two key variables in one composite index.
- The two WHERE conditions are connected by the AND operator.
- At least one of the conditions contains the EQ operator.

If the two conditions in the WHERE statement are reversed, as shown below, the statement still meets all requirements for compound optimization. The order in which the key variables appear does not matter.

```
where frstname eq 'John' and
      lastname eq 'Smith';
```

Now suppose that the conditions in the WHERE statement are joined by the operator OR instead of aND:

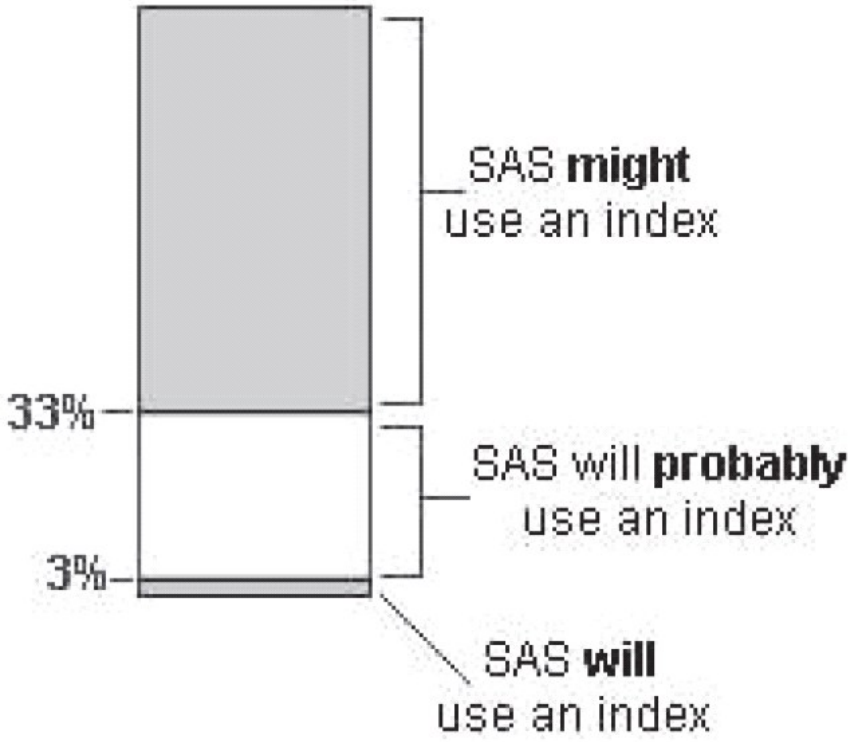
```
where frstname eq 'John' or
      lastname eq 'Smith';
```

These conditions *cannot* be optimized because they are joined by OR but they do not reference the same variable.

Estimating the Number of Observations

Overview

It is more efficient to use indexed access for a small subset and to use sequential access for a large subset. Therefore, after identifying any available indexes and evaluating the conditions in the WHERE expression, SAS estimates the number of observations that will be qualified by the index. Whether SAS uses an index depends on the percentage of observations that are qualified (the size of the subset relative to the size of the data set), as shown below:



- If the subset is *less than 3%* of the data set, direct access is almost certainly more efficient than sequential access, and SAS *will* use an index. In this situation, SAS does *not* go on to compare probable resource usage.
- If the subset is *between 3% and 33%* of the data set, direct access is likely to be more efficient than sequential access, and SAS will *probably* use an index.
- If the subset is *greater than 33%* of the data set, it is less likely that direct access is more efficient than sequential access, and SAS *might or might not* use an index.

When multiple indexes exist, SAS selects the one that appears to produce the *fewest* qualified observations (the *smallest subset)*. SAS does this even when each index returns a subset that is less than 3% of the data set.

Printing Centile Information

To help SAS estimate the number of observations that would be selected by a WHERE expression, each index stores 21 statistics called *cumulative percen tiles*, or *centiles*. Centiles provide information about the distribution of values for the indexed variable.

Understanding the distribution of values in a data set can help you improve the efficiency of WHERE processing in your programs. You can print centile information for an indexed data file by specifying the CENTILES option in either of these places:

- the CONTENTS procedure
- the CONTENTS statement in the DATASETS procedure.

PROC CONTENTS *<options>*; **RUN;**	**PROC DATASETS** *<options>*; **CONTENTS** *<options>*; **QUIT;**

Example

The following SAS program prints centile information for the data set *Company.Organization :*

```
proc contents data=company.organization centiles;
run;
```

Partial output from this program is shown below. As indicated on the left, an index is defined on the variable **`Employee_ID`**. The 21 centile values are listed on the right.

Alphabetic List of Indexes and Attributes							
#	Index	Unique Option	Owned by I C	Update Centiles	Current Update Percent	# of Unique Values	Variables
1	EnnployeeJC	YES	YES	c	0	1049	
							120101
							120152
							120205
							120257
							120310
							120362
							120415
							120467
							120520
							120572
							120625
							120677
							120729
							120782
							120834
							120887
							120939
							120992
							121044
							121097
							99999999

Figure 24.1: Partial PROC CONTENTS Output

The 21 centile values consist of the following:

Position in List	Value Shown in Output Above	Description
1 (first)	*120101*	the *minimum value* of the indexed variable (0% of values are lower than this value)
2-20	*120152 -121097*	each value is greater than or equal to all other values in one of the 19 percentiles that range from the bottom 5% to the bottom 95% of values, in increments of 5%
21 (last)	*99999999*	the *maximum value* of the indexed variable (100% of values are lower than or equal to this value)

Note For information about updating and refreshing centiles for a data file, see the SAS documentation.

Comparing Probable Resource Usage

Overview

Once SAS estimates the number of qualified observations and selects the index that qualifies the fewest observations, SAS

must then determine whether it is faster (more efficient) to satisfy the WHERE expression by using the index or by reading all of the observations sequentially. Specifically, SAS predicts how many *I/O operations* will be required in order to satisfy the WHERE expression for each of the access methods. Then it compares the two resource costs.

Note Remember, if SAS estimates that a subset contains fewer than 3% of the observations in the data set, SAS does not need to estimate resource usage. In this situation, SAS will use the index to process the WHERE statement.

How SAS Compares Resource Usage

To compare resource usage, SAS performs the following steps:

1. SAS predicts how many I/O operations will be required if it uses the index to satisfy the WHERE expression. To do so, SAS positions the index at the first entry that contains a qualified value. In a buffer management simulation that takes into account the current number of available buffers, the RIDs (record identifiers) on that index page are processed, indicating how many I/Os will be required in order to read the observations in the data file.
2. SAS calculates the I/O cost of a sequential pass of the entire data file.
3. SAS compares the two resource costs and determines which access method has a lower cost.

Note If comparing resource costs results in a tie, SAS chooses the index.

Factors That Affect I/O

Several factors affect the number of I/O operations that are required for WHERE processing, including the following:

- subset size relative to data set size
- number of pages in the data tile
- order of the data
- cost to uncompress a compressed tile for a sequential read.

These factors are discussed in more detail below.

Subset Size Relative to Data Set Size

As explained earlier in this chapter, SAS is more likely to use an index to access a small subset of observations. The process of retrieving data with an index is inherently more complicated than sequentially processing the data. This is why using an index requires more I/O operations and CPU time when a large subset is read.

For small subsets, however, the benefit of reading only a few observations outweighs the cost of the complex processing. The smaller the subset, the larger the performance gains. Remember that SAS will use an index If the subset is less than 3% of the data set, and SAS will probably use an index If the subset is between 3% and 33% of the data set.

Number of Pages in the Data File

For a small data file, sequential processing is of ten just as efficient as index processing. If the data file's page count is less than three pages, then sequential access is faster even If the subset is less than 3% of the entire data set.

Note The amount of data that can be transferred to one buffer in a single I/O operation is referred to as page size. To see how many pages are in a data file, use either the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure. For more information about reporting the page size for a data file, see "Controlling Memory Usage" on page 711.

Order of the Data

The order of the data (sort order) affects the number of I/O operations as described below:

Order of the Data	Effect on I/O Operations
observations are *randomly* distributed throughout the data	The observations are located on a larger number of data file pages. An I/O operation is required each time that SAS loads a page. Therefore, the *more random* the data in the data file, the *more*

file	*I/O operations* are needed to use the index.
observations are *sorted* on the indexed variable(s)	The data is ordered more like the index (in ascending value order), and the observations will be located on fewer data file pages. Therefore, the *less random* the data in the data file, *fewer* I/O operations are needed to use the index.

Note In general, sorting the data set by the key variable before indexing will result in greater efficiency. The more ordered the data file is with respect to the key variable, the more efficient the use of the index. If the data file has more than one index, then sorting the data by the most frequently used key variable is most efficient. Sorting the data set results in more efficient WHERE processing even when SAS does not use an index. To learn more about sorting and efficiency, see "Selecting Efficient Sorting Strategies" on page 810.

Cost to Uncompress a Compressed File for a Sequential Read

When SAS reads a compressed data file, SAS automatically uncompresses the observations as they are read into the program data vector. This requires additional CPU resources, but fewer I/O operations are required because there are fewer data set pages. When performing a *sequential* read of a compressed data file, SAS must uncompress *all* observations in the file. However, when using *direct* access, SAS must uncompress only the *qualified* observations. Therefore, the resource cost of uncompressing observations is greater for a sequential read than for direct access.

Note Compressing a file is a process that reduces the number of bytes that are required for representing each observation. By default, a SAS data file is not compressed. For more information about compressing files, see "Controlling Data Storage Space" on page 730.

Other Factors That Affect Resource Usage

Data type and *length* are two other factors that can affect index efficiency.

- *Numeric* key variables typically result in more CPU usage than character key variables, because numeric variables must be converted to formats that can be sequenced when values are read into the index or retrieved from the index. *Character* values are already in a format that can be sequenced.
- When character values contain numerous values, the index structure cannot contain as many values, which might lead to more I/O and search time.

Deciding Whether to Create an Index

In previous sections, you learned how SAS determines whether sequential access or direct access is likely to be most efficient for WHERE processing. You also learned about a variety of factors that you can assess to determine which access method is most efficient. Once you have made your determination, you can use the following guidelines to decide whether it is efficient to create an index.

Guidelines for Deciding Whether to Create an Index

- Minimize the number of indexes to reduce disk storage and update costs. Create indexes only on variables that are often used in queries or (when data cannot be sorted) in BY-group processing.
- Create an index when you intend to retrieve a small subset of observations from a large data file.
- Do not create an index If the data file's page count is less than three pages. It is faster to access the data sequentially.
- Create indexes on variables that are discriminating. Discriminating variables have many different values that precisely identify observations. A WHERE expression that subsets based on a discriminating variable results in a smaller subset than a WHERE expression that references a non-discriminating variable (a variable that has only two values, for example, gender does not make a good variable on which to create an index).
- To reduce the number of I/O operations that are performed when you create an index, first sort the data by the key variable. Then, to improve performance, maintain the data file in sorted order by the key variable.

  Note If you choose not to use an index and the data set is large, it is still more efficient to sort the data set on the variable(s) that are specified in the WHERE statement.

- Consider how often your applications use an index. An index must be used often in order to compensate for the

resources that are used in creating and maintaining it.

- Consider the cost of an index for a data file that is frequently changed.
- When you create an index to process a WHERE expression, do not try to create one index that is used to satisfy all queries.

Consider three sample queries to see how you can apply the guidelines that are listed in the previous section. These queries illustrate the effect of one factor—the size of the subset relative to the size of the data set—on the choice of an access method. For each query, you will learn:

- which access method SAS is likely to select
- whether you could improve performance by creating an index.

Example: Selecting Subsets of Various Sizes from Data Sets of Various Sizes

Suppose you are working with the following two data sets, each of which contains information about a company's orders:

Data Set Name	Pages	Observations
Company.Orders_large	285,500	19,033,380
Company.Orders_small	2	140

You want to create queries to generate three subset detail reports, one for each of the following types of subsets:

- *small* subset from a *large* data set
- *large* subset from a *large* data set
- *small* subset from a *small* data set.

In all three queries, the WHERE expression specifies the variable **Order_Date.** You know that this variable will be used frequently in queries for the company, and that it is a discriminating variable. According to the guidelines in the previous section, these are both criteria for creating an index on the variable. However, there is currently no index defined on this variable in either data set.

Query 1: Small Subset from a Large Data Set

The first report that you want to generate shows all orders in *Company.OrdersJarge* that were made on January 10, 1998. Your query is shown below, along with the subset size that you have estimated:

Query	Subset Size

`data _null_;`
`  set company.orders_large;`
`  where order_date='10JAN1998'd;`
`run;`
	2232 observations (out of l9,033,380) = < .
	02% of the dataset

Because the subset is less than 3% of the entire data set, using an index on **Order_Date** should be more efficient than using sequential access. SAS will use an index for WHERE processing, If an index is available. To improve performance, you should create an index on **Order_Date** before running this program.

Query 2: Large Subset from a Large Data Set

The second report shows all orders in *Company.OrdersJarge* that were made before January 1, 2000. Your query and the estimated subset size are shown below:

Query	Subset Size
`data _null_;` `  set company.orders_large;`	*12,752,365* observations (out of l9,033,380) =approximately *67%* of the data set

```
   where order_date<'01JAN2000'd;
run;
```

Because the subset is more than 33% of the entire data set, using the index is probably *less* efficient than using sequential access. SAS will probably not use the index for WHERE processing.

Query 3: Small Subset from a Small Data Set

The third report shows all orders in the smaller data set *Company.Orders_small* that were made on June 30, 1998. Your query and the estimated subset size are shown below:

Query	Subset Size

`data _null_;`
`set company.orders_small;`
`where order_date'30JUN1998'd;`
`run;`
	2 observations (out of 140) =*<2%* of the data set

Because the subset is less than 3% of the entire data set, SAS will use the index for WHERE processing. However, the data file's page count is less than three pages, so it is more efficient to use sequential access. In this situation, it is best not to create an index.

Using the Options IDXWHERE= and IDXNAME= to Control Index Usage

In most situations, it is best to let SAS determine whether or not to use an index for WHERE processing. However, sometimes you might want to control whether or not SAS uses an existing index. For example, if you know that your query will select a large subset and that indexed access will therefore not be efficient, you can tell SAS to ignore any index and to satisfy the conditions of the WHERE expression with a sequential search of the data set. Or, if your query will select a small subset and there are multiple available indexes, you can make sure that SAS uses a particular index to process your WHERE statement. Finally, you might want to force SAS to use (or not use) an index when you are benchmarking.

You should be familiar with the data set options *IDXWHERE*= and *IDXNAME*=, which you can use to control index usage:

Option	Action
IDXWHERE=	specifies whether or not SAS should use an index to process the WHERE expression, no matter which access method SAS estimates is faster. You cannot use IDXWHERE= to override the use of an index for processing a BY statement.
IDXNAME=	causes SAS to use a specific index.

Caution You can use *either* IDXWHERE= or IDXNAME=, but not both at the same time, to control index usage.

For more information about the IDXWHERE= and IDXNAME= data set options, see "Creating and Managing Indexes Using PROC SQL" on page 238.

Specifying MSGLEVEL=I to Determine Whether SAS Is Using an Index

To determine whether SAS is using an index to process a WHERE expression, you can specify/as the value of the MSGLEVEL= system option. Using MSGLEVEL=I causes SAS to display information about index usage in the SAS log.

Note To make the most efficient use of resources, use MSGLEVEL=I only for debugging and for verifying index usage.

Note For more information about the MSGLEVEL= system option, see "Creating Samples and Indexes" on page 470 or "Creating and Managing Indexes Using PROC SQL" on page 238.

Example: Using IDXWHERE=NO to Prevent Index Usage

Suppose you write the following query, which lists all employees who work in the Sales department of a company:

```
proc print data=company.organization;
   where department='Sales';
```

```
run;
```

Now suppose an index is defined on the variable **Department** in the data set *Company.Organization*. You know that **Department** has the value *Sales* in 65% of the observations, so it is *not* efficient for SAS to use an index for WHERE processing. To ensure that SAS does not use an index, specify IDXWHERE=NO after the data set name. At the beginning of the program, you can also add an OPTIONS statement that specifies MSGLEVEL=I to display a message about index usage in the SAS log. The revised program is shown below:

```
options msglevel=i;
proc print data=company.organization (idxwhere=no);
   where department='Sales';
run;
```

When you run this program, the SAS log indicates that the index was *not* used for processing.

Table 24.1: SAS Log

```
INFO: Data set option (IDXWHERE=NO) forced a sequential pass of the data rather than use of an
index for where-clause processing.
```

Comparing Procedures That Produce Detail Reports

Overview

When you want to use a query to produce a detail report, you can choose between the *PRINT procedure* and the *SQL procedure'*.

Procedure	Description
PROC PRINT	▪ produces data listings quickly ▪ can supply titles, footnotes, and column sums
PROC SQL	▪ combines SQL and SAS features such as formats ▪ can manipulate data and create a SAS data set in the same step that creates the report ▪ can produce column and row statistics ▪ does not offer as much control over output as PROC PRINT

To perform a particular task, a single-purpose tool like PROC PRINT generally uses fewer computer resources than a multi-purpose tool like PROC SQL. However, PROC SQL of ten requires fewer and shorter statements to achieve the results that you want.

To illustrate the differences in resource usage between PROC PRINT and PROC SQL, consider some sample queries.

Example: Using PROC PRINT and PROC SQL to Create Detail Reports

Suppose you are working with the data set *Company.Products* and that you want to generate four types of detail reports:

- simple detail report
- subset detail report
- sorted detail report
- sorted subset detail report.

For the first three reports, the PROC PRINT program is likely to use fewer resources than the PROC SQL program. For the last report, the resource usage for the two programs is likely to be about the same.

Report 1: Simple Detail Report

The simple detail report lists the product ID, product name, and supplier name for all products. The PROC PRINT program and PROC SQL program for producing this report are shown below:

PROC PRINT	PROC SQL

proc print data=company.products;
  var product_id product_name
  supplier_name;
run;
	proc sql;
	  select product_id product_name
	  supplier_name
	  from company.products;
	quit;

In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

Report 2: Subset Detail Report

The subset detail report lists the product ID, product name, and supplier name for all products that come from Sweden (SE). The PROC PRINT program and PROC SQL program for producing this report are shown below:

PROC PRINT	PROC SQL

proc print data=company.products;
  var product_id product_name
  supplier_name;
  where supplier_country='SE';
run;
	proc sql;
	  select product_id product_name
	  supplier_name
	  from company.products
	  where supplier_country='SE';
	quit;

Both steps use WHERE processing to subset the data. In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

Report 3: Sorted Detail Report

The sorted detail report lists the product ID, product name, and supplier name for all products, with observations sorted by the supplier country. The PROC PRINT program and PROC SQL program for producing this report are shown below:

PROC PRINT	PROC SQL

proc sort data=company.products
  out=product;
  by supplier_country;
run;
 	
proc print data=product;
  var product_id product_name
  supplier_name;
run;
	proc sql;
	  select product_id product_name
	  supplier_name
	  from company.products
	  order by supplier_country;
	quit;

To sort the data, a PROC SORT step has been added to the PROC PRINT program, and an ORDER BY clause has been added to the PROC SQL program. In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

Report 4: Sorted Subset Detail Report

The sorted subset detail report lists the product ID, product name, and supplier name for all products that come from Sweden (SE), with observations sorted by the supplier name. The PROC PRINT program and PROC SQL program for producing this report are shown below:

PROC PRINT	PROC SQL

proc sort data=company.products
  (keep=Product_ID Product_Name
  Supplier_Name Supplier_Country)
  out=product;
	proc sql;
	  select product_id product_name
	  supplier_name
	  from company.products

```
    where supplier_country='SE';
  by supplier_name;
run;

proc print data=product;
  var product_id product_name
      supplier_name;
run;
```

```
     where supplier_country='SE'
     order by supplier_name;
quit;
```

To sort the data, a PROC SORT step has been added to the PROC PRINT program. The PROC SORT step uses the KEEP= option to subset the observations, which improves efficiency. The PROC SQL step uses an ORDER BY clause for sorting and a WHERE clause for subsetting. In this situation, the CPU and memory usage for the PROC PRINT program and the PROC SQL program are about the same.

Comparing Tools for Summarizing Data

Overview

SAS provides a variety of tools for summarizing data. These summarization tools generate similar but not identical output, and they vary in efficiency. This section discusses the relative efficiency of the following summarization tools.

Note Throughout this section, all references to the MEANS procedure apply also to the SUMMARY procedure.

Tool	Description
MEANS procedure or SUMMARY procedure	▪ computes descriptive statistics for numeric variables ▪ can produce a printed report and create an output data set
TABULATE procedure	▪ produces descriptive statistics in a tabular format ▪ can produce l-,2-, or 3-dimensional tables with descriptive statistics ▪ can also create an output data set
REPORT procedure	▪ combines features of the PRINT, MEANS, and TABULATE procedures with features of the DATA step in a single report-writing tool that can produce a variety of reports ▪ can also create an output data set
SQL procedure	▪ computes descriptive statistics for one or more SAS data sets or DBMS tables ▪ can produce a printed report or create a SAS data set
DATA step	▪ can produce a printed report ▪ can also create an output data set

Note You can also use the FREQ procedure and the UNIVARIATE procedure to generate summary data and create summary reports, but these procedures are not covered in this chapter. For more information about any of these summarization tools, see the SAS documentation.

You can use these tools to summarize data at the following levels:

Level of Summarization	Tools
entire data set	any of the above
one class variable	any of the above To group the data, PROC SQL uses the GROUP BY statement, and the DATA step uses the BY statement.
one or more combinations of class variables	▪ PROC MEANS (or PROC SUMMARY) ▪ PROC TABULATE

Comparing Resource Usage across Summarization Tools

When summarizing data for one or more class variables, the tools in each of the following groups are similar in resource usage:

- *PROCMEANS (or PROCSUMMARY), PROCREPORT, and PROC TABULATE*
- *PROC SQL* and the *DATA* step with PROC SORT.

However, the relative efficiency of the two groups of tools can vary based on the number of values of the CLASS variables. You need to test the techniques with your data.

Comparative Example: Displaying Summary Statistics for One Class Variable

Overview

Suppose you want to summarize the data set *Retail.Orders* by calculating the average quantity of products sold for each order type (each value of the class variable **`Order_Type).`** You can use the following techniques to produce the summary report:

1. PROCMEANS
2. PROCREPORT
3. PROC SORT and a DATA step
4. PROC SQL
5. PROCTABULATE.

The following programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for summarizing data for one class variable.

Programming Techniques

---

❶ PROC MEANS

This PROC MEANS step creates an output report that displays the mean quantity of products sold (the analysis variable **`Quantity)`** for each order type (the class variable **`Order_Type).`**

```
proc means data=retail.orders
     (keep=order_type quantity)
     mean maxdec=2;
   class order_type;
   var quantity;
run;
```

---

❷ PROC REPORT

This PROC REPORT step creates an output report that displays the mean quantity of products sold (the analysis variable **`Quantity)`** for each order type (the class variable **`Order_Type).`**

```
proc report data=retail.orders
     (keep=order_type quantity);
   column order_type quantity;
   define order_type / group width=13
          'Order Type';
   define quantity / mean format=5.2
          'Average Quantity'
          width=8;
run;
```

❸ PROC SORT and a DATA Step

This program uses a PROC SORT step and a DATA step to create an output report. The PROC SORT step specifies the variables to be included in the report, sorts the observations by the values of the variable **Order_Type**, and generates the temporary output data set *Orders*. The DATA step calculates the mean quantity of products sold (the analysis variable **Quantity)** for each order type (the class variable **Order_Type)** and displays these values in the temporary output report.

```
proc sort data=retail.orders
    (keep=order_type quantity)
    out=orders;
  by order_type;
run;

data _null_;
   set orders;
   by order_type;
   format average_order 5.2;
   if first.order_type then do;
      num=0;
      sum=0;
   end;
   num+1;
   sum+quantity;
   if last.order_type then do;
      average_order=sum / num;
      file print;
      put @5 'Order Type' @20 'Average Order';
      put;
      put @13 Order_type 1. @27 Average_order 5.2;
   end;
run;
```

❹ PROC SQL

This PROC SQL step creates an output report that displays the mean quantity of products sold (the analysis variable **Quantity)** for each order type (the class variable **Order_Type).**

```
proc sql;
   select order_type,
     avg(quantity) label='Average Order'
                   format=5.2
     from retail.orders
     group by order_type;
quit;
```

❺ PROC TABULATE

This PROC TABULATE step creates an output report that displays the mean quantity of products sold (the analysis variable **Quantity)** for each order type (the class variable **Order_Type).**

```
proc tabulate data=retail.orders
     (keep=order_type quantity)
     format=comma8.2;
   class order_type;
   var quantity;
   table order_type, quantity*mean;
run;
```

General Recommendations

- When summarizing data for one class variable, test with your data to determine which summarization tools are most efficient to use.

Using PROC MEANS to Display Summary Statistics for Combinations of Class Variables

To produce summary statistics for *combinations of class variables*, you can use PROC MEANS in the following ways. These techniques differ in resource usage.

Combinations of Class Variables	Technique	Example

all-possible combinations:
c
b
*b*c*
a
*a*c*
*a*b*
*a*b*c*
specific combinations:
*a*b* and *a*c*
	basic PROC MEANS step
	TYPES statement in
	PROC MEANS
		`proc means data=lib.dataset mean;`
		`  class a b c;`
		`  var salary;`
		`  output out=summary1`
		`  mean=average;`
		`run;`
		`proc means data=lib.dataset mean;`
		`  class a b c;`
		`  var salary;`
		`  types a*b a*c;`
		`  output out=summary2`
		`  mean=average;`
		`run;`

specific combinations:
*a*b* and *a*c*
	NWAY option in multiple PROC MEANS steps	`proc means data=lib.dataset nway;`
		`  class a b;`
		`  var salary;`
		`  output out=summary3a`
		`  mean=average;`
		`run;`
		
		`proc means data=lib.dataset nway;`
		`  class a c;`
		`  var salary;`
		`  output out=summary3b`
		`  mean=average;`
		`run;`

specific combinations: *a*b* and *a*c*	*WHERE= option* in the OUTPUT statement in PROC MEANS	`proc means data=lib.dataset;`
		`  class a b c;`
		`  var salary;`
		`  output out=summary4`
		`  (where=(_type_ in (5,3)))`
		`  n=employees`
		`  mean=average;`
		`run;`

Comparing Resource Usage across Three Techniques for Using PROC MEANS

The three techniques for summarizing data for *specific combinations* of class variables (all but the basic PROC MEANS step) differ in resource usage as follows:

- The *TYPES statement in a PROC MEANS step* uses the fewest resources.
- A program that contains the *NWAY option in multiple PROC MEANS steps* uses the most resources because SAS must read the data set separately for each PROC MEANS step.
- The *WHERE= data set option in a PROC MEANS step* uses more resources than the TYPES statement in PROC MEANS because SAS must calculate all possible combinations of class variables before subsetting. However, the WHERE= data set option in PROC MEANS uses fewer resources than the NWAY option in multiple PROC MEANS steps.

We will learn how to use a basic PROC MEANS step and the three other techniques that are listed above.

Using a Basic PROC MEANS Step to Combine Ail Class Variables

By default, PROC MEANS (or PROC SUMMARY) creates the following:

- an output *report* that groups data and displays summary statistics for the combination of *all the class variables*
- an output *data set* that groups data and displays summary statistics for *all possible combinations* of the *n* class variables (from 1-way to *n*-way), as well as for the entire data set.

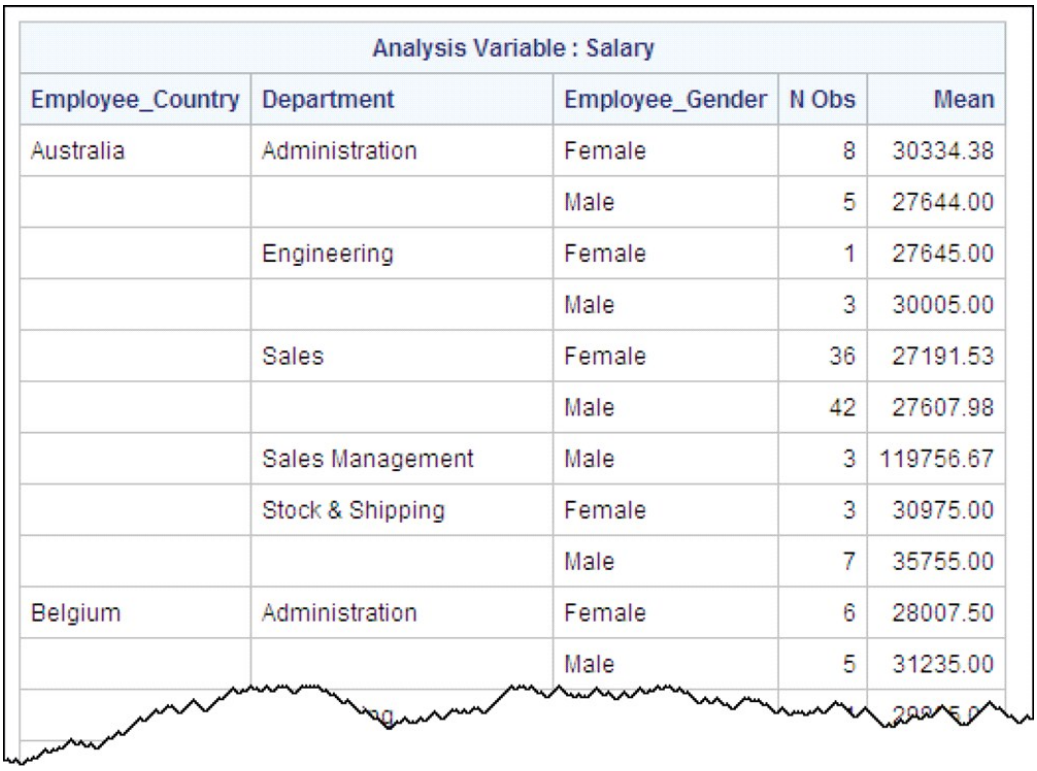
Example: Displaying Summary Statistics for AII Combinations of the Class Variables

Suppose you want to calculate average employee salaries and to group results for the combination of the three class variables **`Employee_Country, Department`**, and **`Employee_Gender`**.

The following PROC MEANS program creates both a report and a SAS data set:

```
proc means data=company.organization_dim mean;
   class employee_country department
        employee_gender;
   var salary;
   output out=summary mean=average;
run;
```

The report groups data and displays summary statistics for the combination of the three class variables. A partial view of the output report is shown below:



Analysis Variable : Salary

Employee_Country	Department	Employee_Gender	N Obs	Mean
Australia	Administration	Female	8	30334.38
		Male	5	27644.00
	Engineering	Female	1	27645.00
		Male	3	30005.00
	Sales	Female	36	27191.53
		Male	42	27607.98
	Sales Management	Male	3	119756.67
	Stock & Shipping	Female	3	30975.00
		Male	7	35755.00
Belgium	Administration	Female	6	28007.50
		Male	5	31235.00

The output data set groups data and displays summary statistics for both of the following:

- all possible combinations (1-way, 2-way, and 3-way) of the three class variables:
  - **`Employee_Gender`**
  - **`Department`**

- **Department** and **Employee_Gender**
- **Employee_Country**
- **Employee_Country** and **Employee_Gender**
- **Employee_Country** and **Department**
- **Employee_Country** and **Department** and **Employee_Gender**

- the entire data set.

A partial view of the output data set is shown below:

obs	Employee_Country	Department	Employee_Gender	_TYPE_	_FREQ_	average
l				0	1048	$33,807
2			Female	1	473	$32,611
3			Male	1	575	$34,791
4		Accounts		2	17	$40,026
5		Accounts Management		2	9	$44,131
6		Administration		2	121	$30,367
7		Concession Management		2	11	$33,839
8		Engineering		2	31	$29,815
9		Executives		2	4	$288,333
10		Group Financials		2	3	$44,022

Understanding Types

Each combination of class variables that is used to calculate and group statistics for PROC MEANS is called a *type*.

For example, the following basic PROC MEANS step specifies the three class variables **a, b**, and **c:**

```
proc means data=lib.dataset mean;
   class a b c;
   var salary;
   output out=summary1
          mean=average;
run;
```

This PROC MEANS step generates seven possible types (combinations of the three variables):

Variables Combined	Dimension
c	1-way
b	1-way
b * c	2-way
a	1-way
a * c	2-way
a * b	2-way
a 1 b * c	3-way

SAS uses the ***_TYPE_*** variable to specify the combination of class variables that PROC MEANS uses to calculate the statistics for each observation in the output data set. The **_TYPE_** variable has a unique value for each combination. SAS always combines the class variables in a particular sequence, based on the order in which they are listed in the CLASS statement, in order to assign the **_TYPE_** values. For example, for each of the seven types (seven possible combinations of three class variables) shown above, SAS assigns a value to **_TYPE_** as follows:

TYPE_ Value	Description of Combination	Variables Combined	Dimension
1	*rightmost variable only*	c	1-way
2	*middle variable only*	b	1-way
3	*rightmost variable and middle variable*	b * c	2-way
4	*leftmost variable*	a	1-way
5	*leftmost variable and rightmost variable*	a * c	2-way
6	*leftmost variable and middle variable*	a * b	2-way
7	*rightmost variable and middle variable and leftmost variable*	a * b * c	3-way

As the number of class variables increases, so does the number of types. However, the *highest* _**TYPE**_ value (7, in this example) always indicates the combination of *all* class variables.

SAS includes the _**TYPE**_ variable in the output data set that is generated by PROC MEANS. In the output from the basic PROC MEANS step that was shown in the previous section, you can see that the observations are listed in order of increasing values of the **TYPE** variable:

obs	Employee_Country	Department	Employee_Gender	_TYPE_	_FREQ_	average
I				0	1048	$33,807
2			Female	1	473	$32,611
3			Male	1	575	$34,791
4		Accounts		2	17	$40,026
5		Accounts Management		2	9	$44,131
6		Administration		2	121	$30,367
7		Concession Management		2	11	$33,839
8		Engineering		2	31	$29,815
9		Executives		2	4	$288,333
10		Group Financials		2	3	$44,022

The first observation in the output data set has a_**TYPE**_ value of *0*, which indicates that the statistics are generated for the *entire data set*.

Note SAS calculates the _**TYPE**_ variable even ifno output data set is requested.

By default, the output *data set* that is generated by PROC MEANS contains a separate observation for each unique combination of values of the class variables for each type. Each unique combination of values within a type is called a *level* of that type. In the output data set linked above (for example, there are 17 levels for type 2,17 observations that have a _**TYPE**_ value of 2).

The output *report* that is generated by the basic PROC MEANS step contains only the observations that represent a combination of *all* of the class variables (the observations for which _**TYPE**_=7). The _**TYPE**_ variable is not displayed in the report.

Using the TYPES Statement in PROC MEANS to Combine Class Variables

You can use the TYPES statement in PROC MEANS to specify which combinations of the class variables are used for grouping data and for calculating statistics. The CLASS statement is required in order to use the TYPES statement.

Table 24.2: SAS Log

General form, TYPES statement:

TYPES *request(s)*;

where

request(s)

specifies the combination(s) of class variables that PROC MEANS uses to create the types. A request includes one of the following:

- one class variable name
- several class variable names separated by asterisks
- ( ) to request the combination of all variables(_**TYPE**_=0).

To request combinations of class variables more concisely, you can use a grouping syntax by placing parentheses around several variables and joining other variables or variable combinations. The following examples of TYPES statements illustrate the use of grouping syntax:

Example with Grouping Syntax	Equivalent Example without Grouping Syntax
`types a*(b c);`	`types a*b a*c;`
`types (a b)*(c d);`	`types a*c a*d b*c b*d;`
`types (a b c)*d;`	`types a*d b*d c*d;`
`types () a*(b c);`	`types a*b*c a*b a*c;`

Example: Using the TYPES Statement in PROC MEANS

Suppose you want to calculate average employee salaries, as in the previous example. This time, you want to group results for the two combinations of class variables shown below:

- **Employee_Country** and **Department**
- **Employee_Country** and **Employee_Gender**.

To do this, you can add a TYPES statement to the PROC MEANS step:

```
proc means data=company.organization_dim mean;
   class employee_country department
         employee_gender;
   var salary;
   types employee_country*department
         employee_country*employee_gender;
   output out=summary mean=average;
run;
```

This PROC MEANS step generates both an output report and an output data set. The report, shown below, has a separate table for each of the two combinations that were specified in the TYPES statement:

Analysis Variable : Salary Annual Salary

Employee Country	Employee Gender	N Obs	Mean
Australia	Female	48	27961.25
	Male	60	33288.75
Belgium	Female	27	32342.22
	Male	33	29754.39
Germany	Female	45	34060.00
	Male	53	27361.70
Denmark	Female	23	27856.52
	Male	28	34693.75
Spain	Female	29	33353.10
	Male	37	29323.11
France	Female	44	28831.70
	Male	54	34143.24
United Kingdom	Female	50	28239.70
	Male	59	32507.88
Italy	Female	34	29634.85
	Male	43	30980.23
Netherlands	Female	30	30834.83
	Male	35	31160.00
United States	Female	143	38152.45
	Male	173	42395.00

Analysis Variable : Salary Annual Salary

Employee Country	Department	N Obs	Mean
Australia	Administration	13	29299.62
	Engineering	4	29415.00
	Sales	78	27415.77
	Sales Management	3	119756.67
	Stock & Shipping	10	34321.00
Belgium	Administration	11	29474.55

The output data set summarizes and reports data for only the combinations (types) that are specified in the TYPES statement. A partial view of the output data set is shown below:

Obs	Employas_Country	Department	Employee_candor	_TYPE_	_FREQ_	average
1	AU		F	5	48	$27,961
2	AU		M	5	60	$33,289
3	BE		F	5	27	$32,342
4	BE		M	5	33	$29,754
5	DE		F	5	45	$34,060
6	DE		M	5	53	$27,362
7	DK		F	5	23	$27,857
8	DK		M	5	28	$34,694
9	ES		F	5	29	$33,353
1O	ES		M	5	37	$29,323

Using the NWAY Option in PROC MEANS to Combine Class Variables

Another way to specify a combination of class variables is to use the NWAY option in PROCMEANS:

General form, NWAY option in the PROC MEANS statement:

PROC MEANS NWAY;

where

NWAY

specifies that the output will contain statistics for the combination of all specified class variables (the observations that have the highest _**TYPE**_ value).

The NWAY option enables you to generate summary statistics for *one* particular combination of class variables—*all* of the class variables—in a single PROC MEANS step. Therefore, to generate statistics for different combinations of class variables, you can specify a separate PROC MEANS step that contains the NWAY option for *each* combination.

Example: Using the NWAY Option in Multiple PROC MEANS Steps

Suppose you want to calculate average employee salaries and to group results for the following combinations of class variables:

- **`Employee_Country`** and **Department**
- **`Employee_Country`** and **`Employee_Gender`**.

You can use two PROC MEANS steps, each containing the NWAY option, as shown below. The first PROC MEANS step generates statistics for the first combination of class variables, and the second PROC MEANS steps generates statistics for the second combination of class variables.

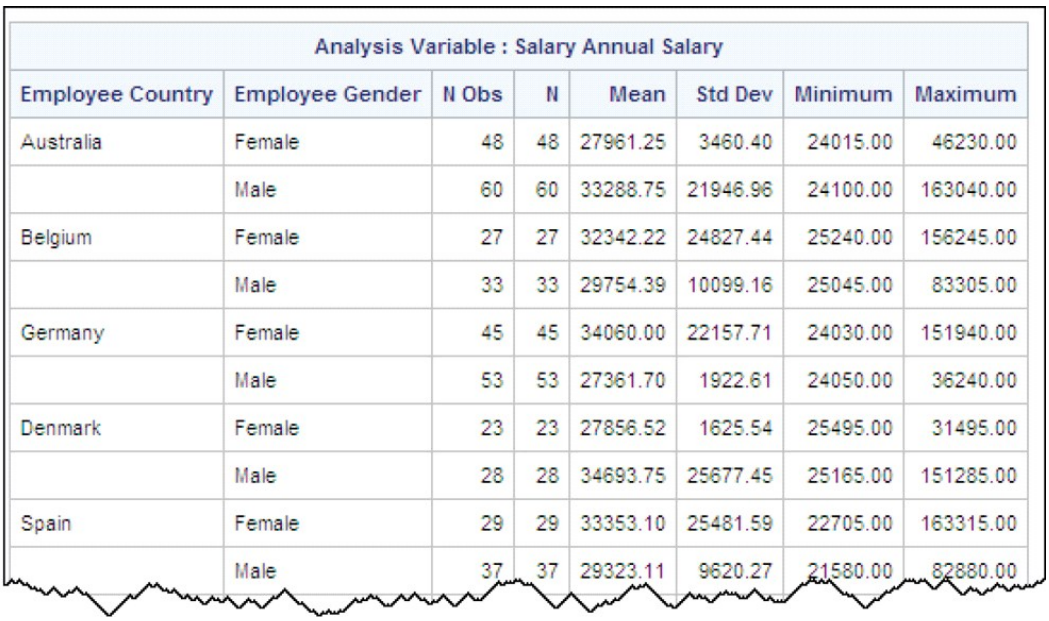
```
proc means data=company.organization_dim nway;
   class employee_country department;
   var salary;
   output out=summary1
          n=employees mean=average;
run;
proc means data=company.organization nway;
   class employee_country employee_gender;
   var salary;
   output out=summary2
          n=employees
          mean=average;
run;
```

When processing this program, SAS must read the data set once for each PROC MEANS step.

This program generates an output report and two output data sets. The report, shown in part below, has a separate table for each PROC MEANS step:

Analysis Variable : Salary Annual Salary

Employee Country	Department	N Obs	N	Mean	Std Dev	Minimum	Maximum
Australia	Administration	13	13	29299.62	5325.55	26495.00	46230.00
	Engineering	4	4	29415.00	1886.72	27645.00	31670.00
	Sales	78	78	27415.77	2120.26	24015.00	36605.00
	Sales Management	3	3	119756.67	38831.75	87975.00	163040.00
	Stock & Shipping	10	10	34321.00	11702.89	26160.00	60980.00
Belgium	Administration	11	11	29474.55	5054.34	25045.00	43280.00
	Engineering	2	2	28532.50	1799.59	27260.00	29805.00
	Sales	45	45	27428.89	1489.26	25240.00	32470.00
	Sales Management	2	2	119775.00	51576.37	83305.00	156245.00
Germany	Administration	13	13	30708.85	7043.84	25705.00	48100.00
		5			3809.44	27250.00	36240.00



Analysis Variable : Salary Annual Salary							
Employee Country	Employee Gender	N Obs	N	Mean	Std Dev	Minimum	Maximum
Australia	Female	48	48	27961.25	3460.40	24015.00	46230.00
	Male	60	60	33288.75	21946.96	24100.00	163040.00
Belgium	Female	27	27	32342.22	24827.44	25240.00	156245.00
	Male	33	33	29754.39	10099.16	25045.00	83305.00
Germany	Female	45	45	34060.00	22157.71	24030.00	151940.00
	Male	53	53	27361.70	1922.61	24050.00	36240.00
Denmark	Female	23	23	27856.52	1625.54	25495.00	31495.00
	Male	28	28	34693.75	25677.45	25165.00	151285.00
Spain	Female	29	29	33353.10	25481.59	22705.00	163315.00
	Male	37	37	29323.11	9620.27	21580.00	82880.00

Each output data set summarizes and reports data for one of the types that are specified in the TYPES statement. A partial view of each output data set is shown below:

Obs	Employee_Country	Department	_TYPE_	_FREQ_	employees	overage
1	Australia	Administration	3	13	13	$29,300
2	Australia	Engineering	3	4	4	$29,415
3	Australia	Sales	3	78	78	$27,416
4	Australia	Sales Management	3	3	3	$119,757
5	Australia	Stock & Shipping	3	10	10	$34,321
6	Belgium	Administration	3	11	11	$29.475
7	Belgium	Engineering	3	2	2	$28,533
8	Belgium	Sales Management	3	45	45	$27,429
9	Belgium	Sales Management	3	2	2	$119,775
10	Germany	Administration	3	13	13	$30,709

Figure 24.2: SAS Data Set Work.Summary 1

Obs	Employee_Country	Employee_Gender	_TYPE_	_FREQ_	employees	overage
1	Australia	Female	3	48	48	$27,961
2	Australia	Male	3	60	60	$33,289
3	Belgium	Female	3	27	27	$32,342
4	Belgium	Male	3	33	33	$29,754
5	Germany	Female	3	45	45	$34,060
6	Germany	Male	3	53	53	$27,362
7	Denmark	Female	3	23	23	$27,857
8	Denmark	Male	3	28	28	$34,694
9	Spain	Female	3	29	29	$33,353
10	Spain	Male	3	37	37	$29,323

Figure 24.3: SAS Data Set Work.Summary 2

Using the WHERE= Option in PROC MEANS to Combine Class Variables

Yet another way to specify a combination of class variables is to use the *WHERE= data set option* in the OUTPUT statement:

General form, WHERE= data set option in a basic OUTPUT statement:

OUTPUT <*OUT=SAS-data-set*> <*output-statistic-specification(s)*> **(WHERE=**

(where-expression-1 <logical-operator where-expression-n>));

where

SAS-data-set

specifies the new output data set as a 1-level or 2-level name.

output-statistic-specification(s)

specifies the statistic(s) to store in the output data set and names one or more variables that contain the statistics.

where-expression

is an arithmetic or logical expression that consists of a sequence of operators, operands, and SAS functions. An operand is a variable, a SAS function, or a constant. An operator is a symbol that requests a comparison, logical operation, or arithmetic calculation. The expression must be enclosed in parentheses.

logical-operator

can be AND, AND NOT, OR, or OR NOT.

When you use the WHERE= option in the OUTPUT statement, SAS must calculate all possible combinations of class variables, and subsetting does not occur until the results are written to output.

Example: Using the WHERE= Option in PROC MEANS

Suppose you want to calculate average employee salaries and to group results for two 2- way combinations of the three class variables **`Employee_Country, Department`**, and **`Employee_Gender.`** All possible combinations of these variables are listed below:

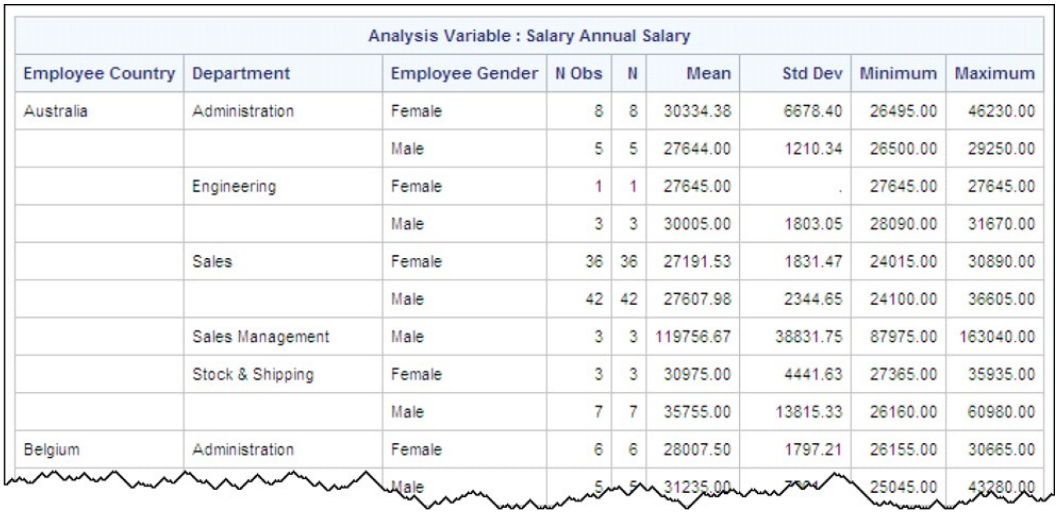
TYPE Value	Variables Combined	Dimension
1	Employee Gender	1-way
2	Department	1-way
3	Department * Employee Gender	2-way
4	Employee_Country	1-way
5	Employee Country * Employee Gender	2-way
6	Employee Country * Department	2-way
7	Employee_Country * Department * Employee_Gender	3-way

To specify the types by **`_TYPE_`** value, you can use the WHERE= option in the OUTPUT statement as shown below:

```
proc means data=company.organization_dim;
   class employee_country department
         employee_gender;
   var salary;
   output out=summary
         (where=(_type_ in (5,6)))
         n=employees
         mean=average;
```

```
run;
```

A partial view of the output report is shown below:



Analysis Variable : Salary Annual Salary

Employee Country	Department	Employee Gender	N Obs	N	Mean	Std Dev	Minimum	Maximum
Australia	Administration	Female	8	8	30334.38	6678.40	26495.00	46230.00
		Male	5	5	27644.00	1210.34	26500.00	29250.00
	Engineering	Female	1	1	27645.00	.	27645.00	27645.00
		Male	3	3	30005.00	1803.05	28090.00	31670.00
	Sales	Female	36	36	27191.53	1831.47	24015.00	30890.00
		Male	42	42	27607.98	2344.65	24100.00	36605.00
	Sales Management	Male	3	3	119756.67	38831.75	87975.00	163040.00
	Stock & Shipping	Female	3	3	30975.00	4441.63	27365.00	35935.00
		Male	7	7	35755.00	13815.33	26160.00	60980.00
Belgium	Administration	Female	6	6	28007.50	1797.21	26155.00	30665.00
		Male	5	5	31235.00	[illegible]	25045.00	43280.00

A partial view of the output data set *Work.Summary* is shown below:

Obs	Employee_Country	Department	Employee_Gender	_TYPE_	_FREQ_	employees	overage
1	Australia		Female	5	48	48	$27,961
2	Australia		Male	5	60	60	$33,289
3	Belgium		Female	5	27	27	$32,342
4	Belgium		Male	5	33	33	$29,754
5	Germany		Female	5	45	45	$34,060
6	Germany		Male	5	53	53	$27,362
7	Denmark		Female	5	23	23	$27,857
8	Denmark		Male	5	28	28	$34,694
9	Spain		Female	5	29	29	$33,353
10	Spain		Male	5	37	37	$29,323

Next, compare the resources that are used by these summarization techniques:

- the TYPES statement in PROC MEANS
- the NWAY option in multiple PROC MEANS steps
- the WHERE= option in PROC MEANS.

Comparative Example: Displaying Summary Statistics for Combinations of Class Variables

Overview

Suppose you want to summarize the data set *Retail.Organization* by calculating average employee salaries for two 3-way combinations off our class variables:

- **`Employee_Country, Department`**, and **`Employee_Gender`**
- **`Department, Section`**, and **`Employee_Gender`**.

You can use the following techniques to produce an output report and one or more output data sets:

1. TYPES Statement in PROC MEANS

2. NWAY Option in Two PROC MEANS Steps
3. WHERE= Option in PROC MEANS

The following programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for summarizing data for combinations of class variables.

Programming Techniques

---

❶ TYPES Statement in PROC MEANS

This program calculates the average employee salary for two 3-way combinations of the class variables **Employee_Country, Department, Employee_Gender**, and **Section.** The TYPES statement specifies the two combinations. The program generates an output report and an output data set named *Summary*.

```
proc means data=retail.organization mean;
   class employee_country department
         employee_gender section;
   var salary;
   types employee_country*department*employee_gender
         department*section*employee_gender;
   output out=summary
          n=employees
          mean=average;
run;
```

---

❷ NWAY Option in Two PROC MEANS Steps

Each of the two PROC MEANS steps in this program calculates the average employee salary for a combination of three of the four class variables **Employee_Country, Department, Employee_Gender**, and **Section.** In each step, the NWAY option specifies that all three variables that are specified in the CLASS statement should be combined. The program generates an output report and two output data sets named *Summaryl* and *Summary2*.

```
proc means data=retail.organization nway;
    class employee_country department
          employee_gender;
    var salary;
    output out=summary1
           n=employees
           mean=average;
run;

proc means data=retail.organization nway;
   class department section
         employee_gender;
   var salary;
   output out=summary2
          n=employees
          mean=average;
run;
```

---

❸ WHERE= Option in PROC MEANS

This program calculates the average employee salary for two 3-way combinations of the class variables **Employee_Country, Department, Employee_Gender**, and **Section.** The WHERE= data set option in the OUTPUT statement specifies the two combinations by their **_TYPE_** values. The program generates an output report and an output data set named *Summary3*.

```
proc means data=retail.organization;
   class employee_country department
         employee_gender section;
   var salary;
   output out=summary3 (where=(_type_ in (7,14)))
          n=employees
          mean=average;
run;
```

General Recommendations

- To summarize data for particular combinations of class variables, use the TYPES statement in PROC MEANS.

Additional Features

The WAYS statement in PROC MEANS provides yet another way to display summary statistics for combinations of class variables. In the WAYS statement, you specify one or more integers that define the number of class variables to combine in order to form all the unique combinations of class variables.

For example, the following program uses the WAYS statement to create summary statistics for the following combinations of the three class variables **`Employee_Country, Department`**, and **`Employee_Gender:`**

- each individual variable (all 1-way combinations)
- all 2-way combinations **`(Employee_Country`** and **`Department, Employee_Country`** and **`Employee_Gender`**, and **`Employee_Gender`** and **`Department)`**.

```
proc means data=company.organization mean;
   class employee_country department
         employee_gender;
   var salary;
   ways 1 2;
   output out=summary
          mean=average;
run;
```

The WAYS statement can be used instead of or in addition to the TYPES statement.

Note For more information about the WAYS statement, see the SAS documentation.

Summary

Using an Index for Efficient WHERE Processing

When processing a WHERE expression, SAS determines whether it is more efficient to access observations in a data set sequentially, by searching through all observations, or directly, by using an index to access specific observations. Using an index to process a WHERE expression might improve performance and is referred to as optimizing the WHERE expression. By deciding whether to create an index, you play a role in determining which access method SAS can use.

In order to decide whether to use an index, you must evaluate the benefits and costs of using an index.

SAS performs a series of steps to determine whether to process a WHERE expression by using an index or by reading all the observations in the data file sequentially.

Identifying Available Indexes

First, SAS determines whether there are any existing indexes that might be used to process the WHERE expression. Specifically, SAS checks the variable in each condition in the WHERE expression to determine whether the variable is either a key variable in a simple index or the first key variable in a composite index. No matter how many indexes are available, SAS can use only one index to process a WHERE expression. So, if multiple indexes are available, SAS must choose between them.

It is most common for SAS to use an index to process just one condition in a WHERE expression. However, in a process called compound optimization, SAS can use a composite index to optimize multiple conditions on multiple variables, which

are joined with a logical operator such as AND.

Identifying Conditions That Can Be Optimized

Second, SAS looks for operators and functions that can be optimized in the WHERE conditions that contain key variables. There are also certain operators and functions that cannot be optimized. For compound optimization, WHERE conditions must meet slightly different criteria in order to be candidates for optimization.

Estimating the Number of Observations

Third, SAS estimates how many observations will be qualified by the index. When multiple indexes exist, SAS selects the one that appears to produce the fewest qualified observations (the smallest subset). Whether or not SAS uses an index depends on the percentage of observations that are qualified (the size of the subset relative to the size of the data set). It is more efficient to use indexed access for a small subset and sequential access for a large subset. If SAS estimates that the number of qualified observations is less than 3% of the data file, SAS automatically uses the index and does not go on to compare probable resource usage.

To help SAS estimate how many observations would be selected by a WHERE expression, each index stores 21 statistics called cumulative percentiles, or centiles. Centiles provide information about the distribution of values for the indexed variable.

Comparing Probable Resource Usage

Fourth, SAS decides whether it is more efficient to satisfy the WHERE expression by using the index or by reading all of the observations sequentially. To make the decision, SAS predicts how many I/O operations will be required in order to satisfy the WHERE expression for each of the access methods, and then compares the two resource costs.

Several factors affect the number of I/O operations that are required for WHERE processing, including the following:

- subset size relative to data set size
- number of pages in the data file
- order of the data
- cost to uncompress a compressed file for a sequential read.

Data type and length are two other factors that affect index efficiency.

Deciding Whether to Create an Index

When you use a WHERE expression to select a subset, you can use specific guidelines to decide whether it is efficient to create an index. Depending on factors such as the size of the subset relative to the size of the data set, you might or might not choose to create an index.

In most situations, it is best to let SAS determine whether or not to use an index for WHERE processing. However, sometimes you might want to control whether or not SAS uses an existing index. You can use either of the data set options IDXWHERE= or IDXNAME=, but not both at the same time, to control index usage. You can specify MSGLEVEL=I to tell SAS to display information about index usage in the SAS log.

Comparing Procedures That Produce Detail Reports

When you want to use a query to produce a detail report, you can choose between the PRINT procedure and the SQL procedure. To perform a particular task, a single-purpose tool like PROC PRINT generally uses fewer computer resources than a multi-purpose tool like PROC SQL. However, PROC SQL often requires fewer and shorter statements to achieve the results that you want.

For detail reports, a PROC PRINT step often, but not always, uses fewer resources than a PROC SQL step:

- PROC PRINT is usually more efficient than PROC SQL for generating a simple detail report, a subset detail report, and a sorted detail report.

- PROC PRINT and PROC SQL will likely have similar resource usage for generating a sorted subset detail report.

Comparing Tools for Summarizing Data

SAS provides a variety of tools for summarizing data, including the MEANS procedure (or SUMMARY procedure), the TABULATE procedure, the REPORT procedure, the SQL procedure, and the DATA step.

If you are summarizing data for one class variable, the tools in each of the following groups are similar in resource usage:

- PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE
- PROC SQL and the DATA step.

However, the relative efficiency of the two groups of tools varies according to the shape of the data.

You can use PROC MEANS in a variety of ways to produce summary statistics for combinations of class variables. Each combination of class variables is called a type.

To summarize data for *all* combinations of class variables, you can use a basic PROC MEANS step (or PROC SUMMARY step). To produce summary statistics for *specific* combinations of class variables, you can use PROC MEANS in the following ways :

- the TYPES statement in a PROC MEANS step
- the NWAY option in multiple PROC MEANS steps
- the WHERE= option in a PROC MEANS step.

These three techniques vary in efficiency; the TYPES statement in PROC MEANS is the most efficient.

You can also use the WAYS statement in PROC MEANS to produce summary statistics for specific combinations of class variables.

Review the related comparative examples:

- "Comparative Example: Displaying Summary Statistics for One Class Variable" on page 884
- "Comparative Example: Displaying Summary Statistics for Combinations of Class Variables" on page 898

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

Quiz

1. Why can using an index reduce the number of I/O operations that are required for accessing a small subset? ?
   a. Using an index requires larger input buffers, which can hold more pages.
   b. The index does not have to be loaded into an input buffer.
   c. The number of observations that SAS has to load into the program data vector (PDV) is decreased.
   d. The number of pages that SAS has to load into input buffers is decreased.
2. You want to select a subset of observations in the data set *Company.Products*, and you have defined a simple index on the variable **`Rating.`** SAS *cannot* use the index to process which of the following WHERE statements? ?
   a. `where rating is missing;`
   b. `where rating=int(rating);`
   c. `where rating between 3.5 and 7.5;`

d. `where rating=5.5;`

3. In which of the following situations is sequential access likely to be more efficient than direct access for WHERE processing? ?

a. The subset contains over 75% of the observations in the data set.

b. The WHERE expression specifies both key variables in a single composite index.

c. The data is sorted on the key variable.

d. The data set is very large.

4. You want to summarize data and group it by one variable. Which of the following tools could not be used? ?

a. The DATA step with BY group processing.

b. The DATA step without BY group processing.

c. PROC SQL with a GROUP BY clause.

d. PROC MEANS with a CLASS statement

5. Which of the following techniques does not summarize data for specific combinations of class variables? ?

a. the NWAY option in multiple PROC MEANS steps

b. the TYPES statement in a PROC MEANS step

c. the WHERE= option in a PROC MEANS step

d. a basic PROC MEANS step

Answers

1. Correct answer: d

When using an index to select a subset, SAS loads only the pages that contain a qualified observation into input buffers. When accessing observations sequentially, SAS must load all observations into input buffers. Loading more pages requires more I/O operations.

2. Correct answer: b

SAS considers using an index to process a WHERE condition that contains one of a specific group of operators and functions. However, SAS will *not* consider using an index for a WHERE condition that contains other elements, such as a function other than TRIM or SUBSTR.

3. Correct answer: a

The size of the subset relative to the size of the data set is an important factor in determining which access method is most efficient. If a subset is large (more than 33% of the data set), it is likely to be more efficient to use sequential access than direct access. Direct access is usually more efficient when you are selecting a small subset (less than 33% of the data set), especially if the data set is large (has a high page count). However, if the data set is very small (less than three pages), using an index is not efficient. The number of key variables specified in a WHERE expression does not determine which access method is most efficient. If the two key variables that are specified are the first two variables in the same index, the WHERE expression is a candidate for compound optimization. Sorting the data also does not determine which access method is most efficient. However, sorting the data before subsetting improves the efficiency of WHERE processing regardless of the access method.

4. Correct answer: b

5. Correct answer: d